that constructor had actually been completed, or any other form of misbehavior over which the developer has no control.

If `Logger::operator<<(const char*)` is designed properly for multithreaded use, then, as of C++11, the previous example has no **data races**, even though the `Logger::Logger(const char* logFilePath)` constructor, i.e., the one used to configure the singleton instance of the logger, is not so designed. That is to say, the implicit **critical section** that is guarded by the compiler includes evaluation of the initializer, which is why a recursive call to initialize a function-scope **static** variable is **undefined behavior** and is likely to result in deadlock; see *Potential Pitfalls — Dangerous recursive initialization* on page 77. ~~Such use~~ of function-scope **static**s~~, however,~~ is not foolproof; see *Potential Pitfalls — Depending on order-of-destruction of local objects after `main` returns* on page 78.

The destruction of **function-scope static** objects is and always has been guaranteed to be safe *provided* (1) no threads are running after returning from `main` and (2) ~~function-scope~~ **static** objects do not depend on each other during destruction; see *Potential Pitfalls — Depending on order-of-destruction of local objects after `main` returns* on page 78.

Use Cases

Meyers Singleton

The guarantees surrounding access across **translation units** to runtime-initialized objects at file or namespace scope are few and weak — especially when that access might occur prior to entering `main`. Consider a library component, `libcomp`, that defines a file-scope **static** singleton, `globalS`, that is initialized at run time:

```
// libcomp.h:
#ifndef INCLUDED_LIBCOMP
#define INCLUDED_LIBCOMP

struct S { /*...*/ };
S& getGlobalS();  // access to global singleton object of type S

#endif
```

```
// libcomp.cpp:
#include <libcomp.h>

static S globalS;
S& getGlobalS() { return globalS; }  // access into this translation unit
```